Interactions

- An interaction is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose.
- A message is a specification of a communication between objects that conveys information with the expectation that activity will ensue.

Context

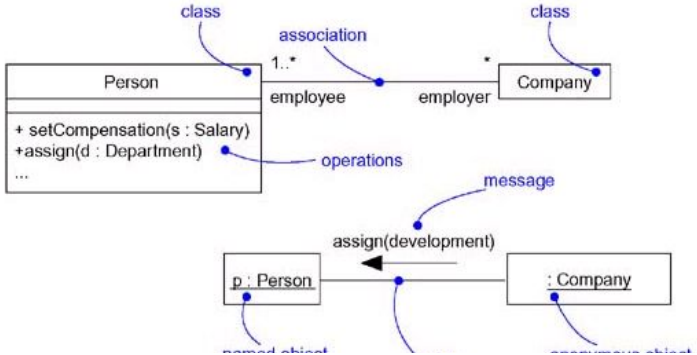
- We can use interactions to visualize, specify, construct, and document the semantics of a class
- We may find an interaction wherever objects are linked to one another.
- We'll find interactions in the collaboration of objects that exist in the context of your system or subsystem.
- We will also find interactions in the context of an operation.
- We might create interactions that show how the attributes of that class collaborate with one another
- Finally, you'll find interactions in the context of a class.

Objects and Roles

- The objects that participate in an interaction are either concrete things or prototypical things.
- As a concrete thing, an object represents something in the real world. For example, p, an instance of the class Person, might denote a particular human
- As a prototypical thing, p might represent any instance of Person.
- Although abstract classes and interfaces, by definition, may not have any direct instances, you may find instances of these things in an interaction
- Such instances do not represent direct instances of the abstract class or of the interface, but may represent, respectively, indirect (or prototypical) instances of any concrete children of the abstract class of some concrete class that realizes that interface.

Links

- A link is a semantic connection among objects. In general, a link is an instance of an association
- Wherever a class has an association to another class, there may be a link between the instances of the two classes. Wherever there is a link between two objects, one object can send a message to the other object
- A link specifies a path along which one object can dispatch a message to another (or the same) object.



Links and Associations

- We can adorn the appropriate end of the link with any of the following standard stereotypes

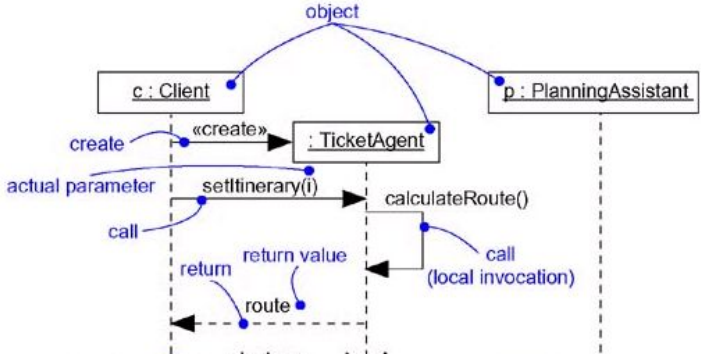
association	Specifies that the corresponding object is visible by association
self	Specifies that the corresponding object is visible because it is the dispatcher of the operation
global	Specifies that the corresponding object is visible because it is in an enclosing scope
local	Specifies that the corresponding object is visible because it is in a local scope
parameter	Specifies that the corresponding object is visible because it is a parameter

Messages

- A message is the specification of a communication among objects that conveys information with the expectation that activity will ensue.
- The receipt of a message instance may be considered an instance of an event.
- When you pass a message, the action that results is an executable statement that forms an abstraction of a computational procedure. An action may result in a change in state.
- In the UML, you can model several kinds of actions

Call	Invokes an operation on an object; an object may send a message to itself, resulting in the local invocation of an operation
Return	Returns a value to the caller
Send	Sends a signal to an object
Create	Creates an object
Destroy	Destroys an object; an object may commit suicide by destroying itself

- The UML provides a visual distinction among these kinds of messages, as follows

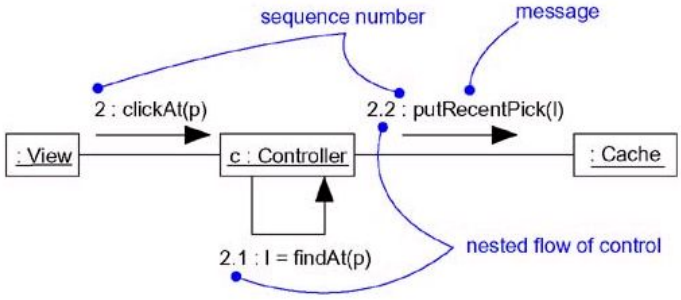


Messages

- When an object calls an operation or sends a signal to another object, you can provide actual parameters to the message.
- Similarly, when an object returns control to another object, you can model the return value.

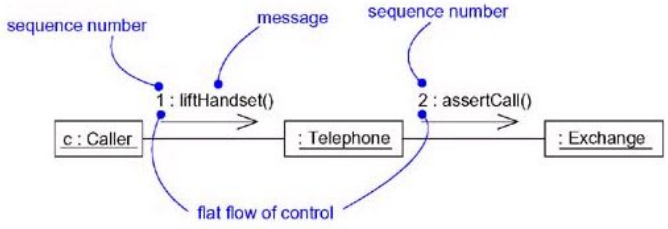
Sequencing

- When an object passes a message to another object the receiving object might in turn send a message to another object, which might send a message to yet a different object, and so on. This stream of messages forms a sequence
- Any sequence must have a beginning; the start of every sequence is rooted in some process or thread.
- Any sequence will continue as long as the process or thread that owns it lives.
- Messages are ordered in sequence by time. To better visualize the sequence of a message, you can explicitly model the order of the message relative to the start of the sequence by prefixing the message with a sequence number set apart by a colon separator
- Most commonly, you can specify a **procedural or nested flow of control**, rendered using a filled



Procedural Sequence

- We can specify a flat flow of control, rendered using a stick arrowhead, to model the nonprocedural progression of control from step to step.



Flat Sequence

- Typically, you'll use flat sequences only when modeling interactions in the context of use cases that involve the system as a whole, together with actors outside the system.
- Such sequences are often flat because control simply progresses from step to step, without any consideration for nested flows of control.

- We'll want to use procedural sequences, because they represent ordinary, nested operation calls of the type you find in most programming languages.

Creation, Modification, and Destruction

- Most of the time, the objects you show participating in an interaction exist for the entire duration of the interaction. However, in some interactions, objects may be created (specified by a create message) and destroyed (specified by a destroy message).
- The same is true of links: the relationships among objects may come and go. To specify if an object or link enters and/or leaves during an interaction, you can attach one of the following constraints to the element:

new	Specifies that the instance or link is created during execution of the enclosing interaction
destroyed	Specifies that the instance or link is destroyed prior to completion of execution of the enclosing interaction
transient	Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution

- Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution
- Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution

Representation

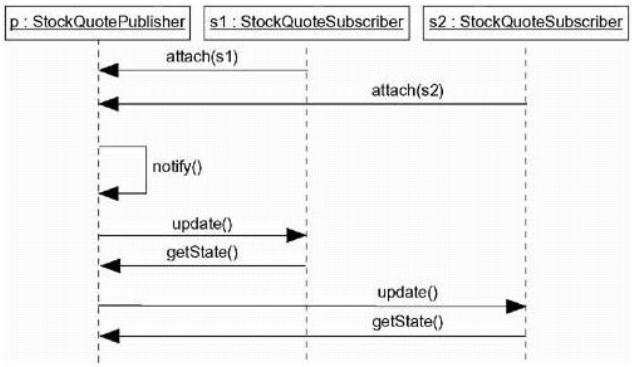
- When you model an interaction, you typically include both objects and messages
- We can visualize those objects and messages involved in an interaction in two ways
    - By emphasizing the time ordering of its messages
    - by emphasizing the structural organization of the objects that send and receive messages.
- In the UML, the first kind of representation is called a sequence diagram
- The second kind of representation is called a collaboration diagram
- Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams
- Sequence diagrams and collaboration diagrams are largely isomorphic
- Sequence diagrams permit you to model the lifeline of an object.

- Collaboration diagrams permit you to model the structural links that may exist among the objects in an interaction.

Common Modeling Techniques

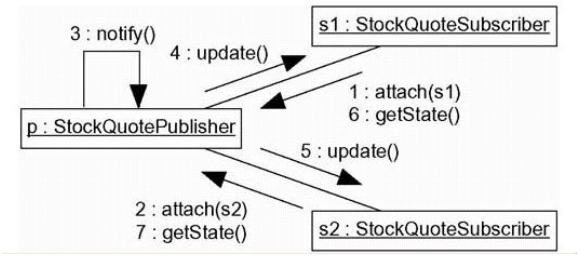
Modeling a Flow of Control

- The most common purpose for which you'll use interactions is to model the flow of control that characterizes the behavior of a system as a whole, including use cases, patterns, mechanisms, and frameworks, or the behavior of a class or an individual operation.
- classes, interfaces, components, nodes, and their relationships model the static aspects of your system
- Interactions model its dynamic aspects of your system.
- To model a flow of control
    - Set the context for the interaction, whether it is the system as a whole, a class, or an individual operation.
    - Set the stage for the interaction by identifying which objects play a role; set their initial properties, including their attribute values, state, and role.
    - If your model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of communication that take place in this interaction. Specify the nature of the links using the UML's standard stereotypes and constraints, as necessary.
    - In time order, specify the messages that pass from object to object. As necessary, distinguish the different kinds of messages; include parameters and return values to convey the necessary detail of this interaction.
    - Also to convey the necessary detail of this interaction, adorn each object at every moment in time with its state and role.
- This figure is an example of a sequence diagram, which emphasizes the time order of messages.



Flow of Control by Time

- This figure is semantically equivalent to the previous one, but it is drawn as a collaboration diagram, which emphasizes the structural organization of the objects.



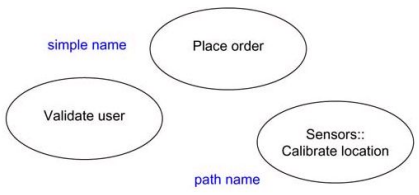
Flow of Control by Organization

Use Cases

- A use case is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor.
- Graphically, a use case is rendered as an ellipse.

Names

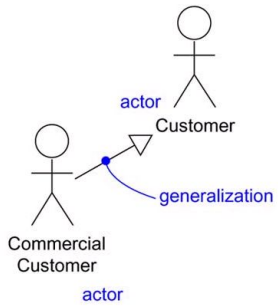
- Every use case must have a name that distinguishes it from other use cases. A name is a textual string.
- That name alone is known as **a simple name**; **a path name** is the use case name prefixed by the name of the package in which that use case lives.
- A use case is typically drawn showing only its name



Simple and Path Names

Use Cases and Actors

- An actor represents a coherent set of roles that users of use cases play when interacting with these
  use cases.
- Typically, an actor represents a role that a human, a hardware device, or another system plays with a system.
- An instance of an actor, therefore, represents an individual interacting with the system in a specific way
- Actors may be connected to use cases only by association
- An association between an actor and a use case indicates that the actor and the use case communicate with one another, each one possibly sending and receiving messages.



Actors

Use Cases and Flow of Events

- A use case describes what a system does but it does not specify how it does it.
- You can specify the behavior of a use case by describing a flow of events in text clearly enough for an outsider to understand it easily
- When you write this flow of events, you should include how and when the use case starts and ends
- When the use case interacts with the actors and what objects are exchanged, and the basic flow and alternative flows of the behavior.

For example, in the context of an ATM system, you might describe the use case ValidateUser in the following way:

Main flow of events:

The use case starts when the system prompts the Customer for a PIN number. The Customer can now enter a PIN number via the keypad. The Customer commits the entry by pressing the Enter button. The system then checks this PIN number to see if it is valid. If the PIN number is valid, the system acknowledges the entry, thus ending the use case.

Exceptional flow of events:

The Customer can cancel a transaction at any time by pressing the Cancel button, thus restarting the use case. No changes are made to the Customer's account.

 Exceptional flow of events:

The Customer can clear a PIN number anytime before committing it and reenter a new PIN number.

 Exceptional flow of events:

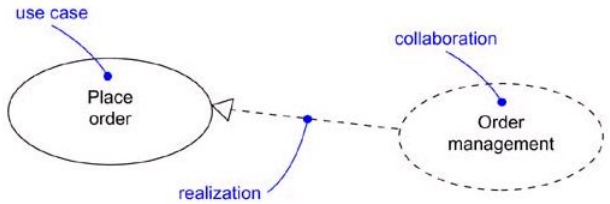
If the Customer enters an invalid PIN number, the use case restarts. If this happens three times in a row, the system cancels the entire transaction, preventing the Customer from interacting with the ATM for 60 seconds

Use Cases and Scenarios

- Typically, we'll first describe the flow of events for a use case in text.
- Typically, we'll use one sequence diagram to specify a use case's main flow, and variations of that diagram to specify a use case's exceptional flows.
- Use case describes a set of sequences, not just a single sequence, and it would be impossible to express all the details of an interesting use case in just one sequence.
- Each sequence is called a **scenario**. A **scenario** is a specific sequence of actions that illustrates behavior. Scenarios are to use cases as instances are to classes, meaning that a scenario is basically one instance of a use case.

Use Cases and Collaborations

- A use case captures the intended behavior of the system you are developing, without having to specify how that behavior is implemented.
- however, you have to implement your use cases, and you do so by creating a society of classes and other elements that work together to implement the behavior of this use case
- This society of elements, including both its static and dynamic structure, is modeled in the UML as a **collaboration.**
- you can explicitly specify the realization of a use case by a collaboration



Use Cases and Collaborations

Organizing Use Cases

- We can organize use cases by grouping them in packages in the same manner in which you can organize classes.
- You can also organize use cases by specifying generalization, include, and extend relationships among them.
- generalization among use cases is rendered as a solid directed line with a large open arrowhead, just like generalization among classes.

- An **include relationship** between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base.
- You use an include relationship to avoid describing the same flow of events several times, by putting the common behavior in a use case of its own
- The include relationship is essentially an example of delegation—you take a set of responsibilities of the system and capture it in one place (the included use case), then let all other parts of the system (other use cases) include the new aggregation of responsibilities whenever they need to use that functionality.
- include followed by the name of the use case you want to include
- You render an include relationship as a dependency, stereotyped as include.

- An **extend relationship** between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case.
- This base use case may be extended only at certain points called, not surprisingly, its extension points
- We use an extend relationship to model the part of a use case the user may see as optional system behavior.
- We may also use an extend relationship to model a separate subflow that is executed only under given conditions.
- Finally, we may use an extend relationship to model several flows that may be inserted at a certain point, governed by explicit interaction with an actor.
- We render an extend relationship as a dependency, stereotyped as extend.

Other Features

- Use cases are classifiers, so they may have attributes and operations that you may render just as for classes.
- You can think of these attributes as the objects inside the use case that you need to describe its outside behavior. Similarly, you can think of these operations as the actions of the system you need to describe a flow of events.
- These objects and operations may be used in your interaction diagrams to specify the behavior of the use case
- As classifiers, you can also attach state machines to use cases
- We can use state machines as yet another way to describe the behavior represented by a use case.

Common Modeling Techniques

Modeling the Behavior of an Element

- The most common thing for which you'll apply use cases is to model the behavior of an element, whether it is the system as a whole, a subsystem, or a class.
- To model the behavior of an element

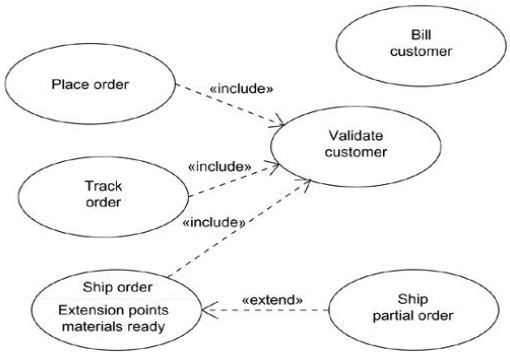
  - Identify the actors that interact with the element. Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element's functions.

  - Organize actors by identifying general and more specialized roles.

  - For each actor, consider the primary ways in which that actor interacts with the element. Consider also interactions that change the state of the element or its environment or that involve a response to some event.

  - Consider also the exceptional ways in which each actor interacts with the element.

  - Organize these behaviors as use cases, applying include and extend relationships to factor common behavior and distinguish exceptional behavior.



Modeling the Behavior of an Element

Use Case Diagram

- A use case diagram is a diagram that shows a set of use cases and actors and their relationships.

Contents

- Use case diagrams commonly contain
    - Use cases
    - Actors
    - Dependency, generalization, and association relationships
- Like all other diagrams, use case diagrams may contain notes and constraints.
- Use case diagrams may also contain packages
- Occasionally, you'll want to place instances of use cases in your diagrams, as well, especially when you want to visualize a specific executing system.

Common Uses

- We apply use case diagrams to model the static use case view of a system. This view primarily supports the behavior of a system
- When you model the static use case view of a system, you'll typically apply use case diagrams in one of two ways.
    - To model the context of a system
    - To model the requirements of a system

Modeling the context of a system involves drawing a line around the whole system and asserting which actors lie outside the system and interact with it.Here, you'll apply use case diagrams to specify the actors and the meaning of their roles.

Modeling the requirements of a system involves specifying what that system should do (from a point of view of outside the system), independent of how that system should do it. Here, you'll apply use case diagrams to specify the desired behavior of the system.

Common Modeling Techniques

Modeling the Context of a System

Given a system—any system—some things will live inside the system, some things will live outside it. For example, in a credit card validation system, you'll find such things as accounts, transactions, and fraud detection agents inside the system. Similarly, you'll find such things as credit card customers and retail institutions outside the system. The things that live inside the system are responsible for carrying out the behavior that those on the outside expect the system to provide. All those things on the outside that interact with the system constitute the system's context. This context defines the environment in which that system lives.

- In the UML, you can model the context of a system with a use case diagram, emphasizing the actors that surround the system.
- To model the context of a system

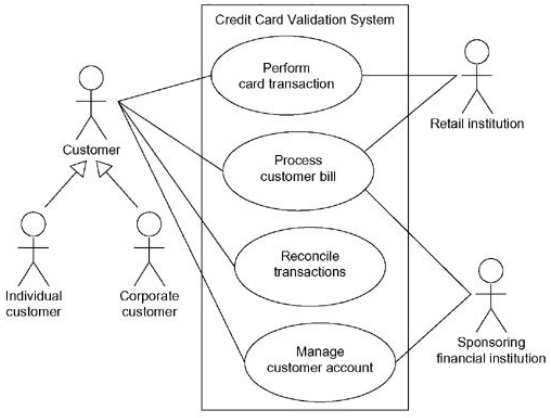
    - Identify the actors that surround the system by considering which groups require help from the system to perform their tasks; which groups are needed to execute the system's functions; which groups interact with external hardware or other software systems; and which groups perform secondary functions for administration and maintenance.

    - Organize actors that are similar to one another in a generalization/specialization hierarchy.

    - Where it aids understandability, provide a stereotype for each such actor.

    - Populate a use case diagram with these actors and specify the paths of communication from each actor to the system's use cases.

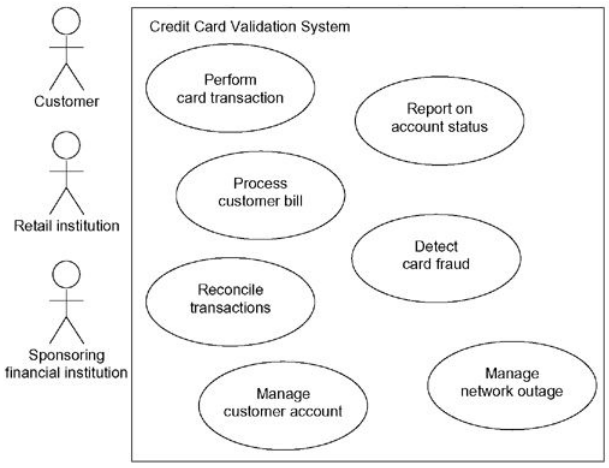
- This same technique applies to modeling the context of a subsystem. A system at one level of abstraction is often a subsystem of a larger system at a higher level of abstraction. Modeling the context of a subsystem is therefore useful when you are building systems of interconnected systems.



Modeling the Context of a System

Modeling the Requirements of a System

- A requirement is a design feature, property, or behavior of a system. When you state a system's requirements, you are asserting a contract, established between those things that lie outside the system and the system itself, which declares what you expect that system to do.
- Requirements can be expressed in various forms, from unstructured text to expressions in a formal language, and everything in between.
- Most, if not all, of a system's functional requirements can be expressed as use cases, and the UML's use case diagrams are essential for managing these requirements.
- To model the requirements of a system,
    - Establish the context of the system by identifying the actors that surround it.
    - For each actor, consider the behavior that each expects or requires the system to provide.
    - Name these common behaviors as use cases.
    - Factor common behavior into new use cases that are used by others; factor variant behavior into new use cases that extend more main line flows.
    - Model these use cases, actors, and their relationships in a use case diagram.
    - Adorn these use cases with notes that assert nonfunctional requirements; you may have to attach some of these to the whole system.
- This same technique applies to modeling the requirements of a subsystem



Modeling the Requirements of a System

Forward and Reverse Engineering

- **Forward engineering** is the process of transforming a model into code through a mapping to an implementation language.
- A use case diagram can be forward engineered to form tests for the element to which it applies.
- Each use case in a use case diagram specifies a flow of events and these flows specify how the element is expected to behave
- To forward engineer a use case diagram,
    - For each use case in the diagram, identify its flow of events and its exceptional flow of events.
    - Depending on how deeply you choose to test, generate a test script for each flow, using the flow's preconditions as the test's initial state and its postconditions as its success criteria.
    - As necessary, generate test scaffolding to represent each actor that interacts with the use case. Actors that push information to the element or are acted on by the element may either be simulated or substituted by its real-world equivalent.
    - Use tools to run these tests each time you release the element to which the use case diagram applies.

- **Reverse engineering** is the process of transforming code into a model through a mapping from a specific implementation language.
- The UML's use case diagrams simply give you a standard and expressive language in which to state what you discover.
- To reverse engineer a use case diagram

- Identify each actor that interacts with the system.
- For each actor, consider the manner in which that actor interacts with the system, changes the state of the system or its environment, or responds to some event.
- Trace the flow of events in the executable system relative to each actor. Start with primary flows and only later consider alternative paths.
- Cluster related flows by declaring a corresponding use case. Consider modeling variants using extend relationships, and consider modeling common flows by applying include relationships.
- Render these actors and use cases in a use case diagram, and establish their relationships.

Activity Diagrams

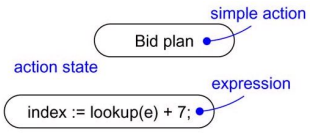
- An activity diagram shows the flow from activity to activity. An is an ongoing nonatomic execution within a state machine.
- Activities ultimately result in some action, which is made up of executable atomic computations that result in a change in state of the system or the return of a value.
- Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression.
- Graphically, an activity diagram is a collection of vertices and arcs.

Contents

- Activity diagrams commonly contain
  - Activity states and action states
  - Transitions
  - Objects
- Like all other diagrams, activity diagrams may contain notes and constraints.

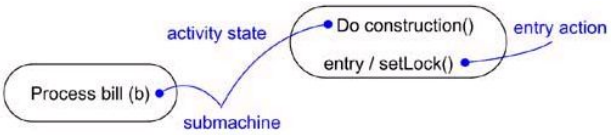
Action States and Activity States

- Executable, atomic computations are called **action states** because they are states of the system, each representing the execution of an action.
- We represent an action state using a lozenge shape (a symbol with horizontal top and bottom and convex sides). Inside that shape, you may write any expression.
- Action states can't be decomposed. Furthermore, action states are atomic, meaning that events may occur, but the work of the action state is not interrupted.
- Finally, the work of an action state is generally considered to take insignificant execution time.



Action States

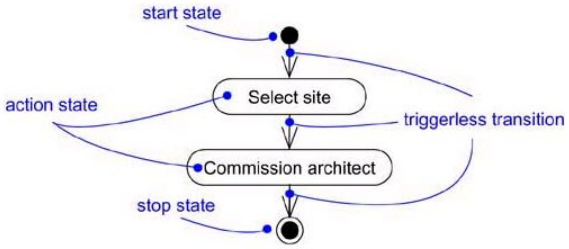
- **activity states** can be further decomposed, their activity being represented by other activity diagrams
- Furthermore, activity states are not atomic, meaning that they may be interrupted and, in general, are considered to take some duration to complete.
- An action state is an activity state that cannot be further decomposed.
- We can think of an activity state as a composite, whose flow of control is made up of other activity states and action states.



Activity States

Transitions

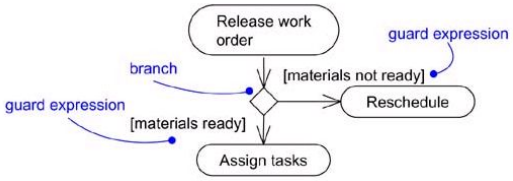
- When the action or activity of a state completes, flow of control passes immediately to the next action or activity state.
- We specify this flow by using transitions to show the path from one action or activity state to the next action or activity state.
- In the UML, you represent a transition as a simple directed line



Triggerless Transitions

Branching

- As in a flowchart, you can include a branch, which specifies alternate paths taken based on some Boolean expression.
- We represent a branch as a diamond. A branch may have one incoming transition and two or more outgoing ones.
- On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch.
- On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch. Across all these outgoing transitions, guards should not overlap (otherwise, the flow of control would be ambiguous), but they should cover all possibilities (otherwise, the flow of control would freeze).
- As a convenience, you can use the keyword else to mark one outgoing transition, representing the path taken if no other guard expression evaluates to true.



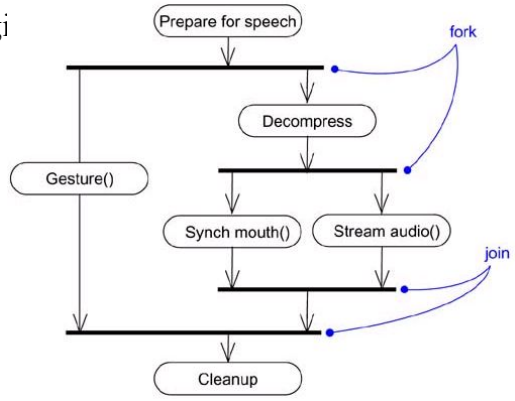
Branching

Forking and Joining

- When we are modeling workflows of business processes—we might encounter flows that are concurrent.

- In the UML, you use a synchronization bar to specify the forking and joining of these parallel flows of control. A synchronization bar is rendered as a thick horizontal or vertical line.

- **Fork** represents the splitting of a single flow of control into two or more concurrent flows of control
- A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control.
- Below the fork, the activities associated with each of these paths continues in parallel.
- Conceptually, the activities of each of these flows are truly concurrent, although, in a running system, these flows may be either truly concurrent or sequential yet interleaved, thus gi :y.



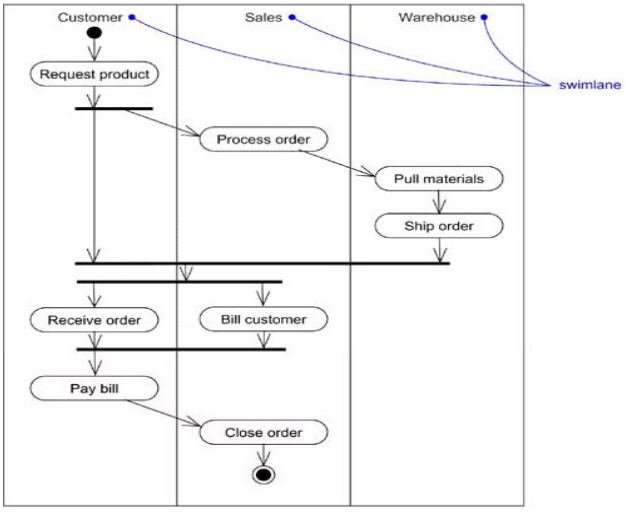
Forking and Joining

- **A Join** represents the synchronization of two or more concurrent flows of control.
- A join may have two or more incoming transitions and one outgoing transition.
- Above the join, the activities associated with each of these paths continues in parallel.
- At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join.

Swimlanes

- We'll find it useful, especially when you are modeling workflows of business processes, to partition the activity states on an activity diagram into groups, each group representing the business organization responsible for those activities.
- In the UML, each group is called a swimlane because, visually, each group is divided from its neighbor by a vertical solid line
- A swimlane specifies a locus of activities
- Each swimlane has a name unique within its diagram.

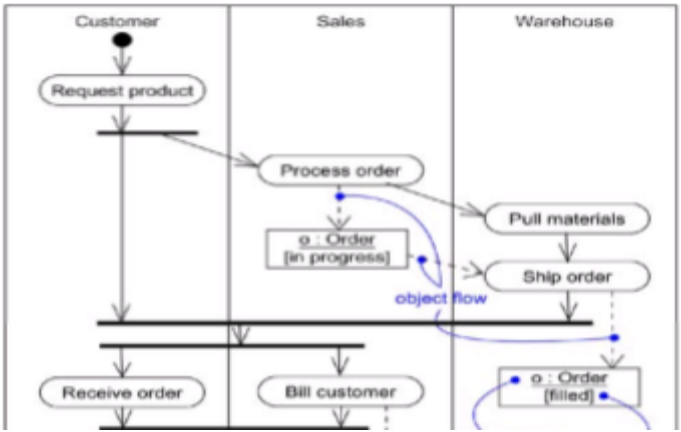
- Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram, and each swimlane may eventually be implemented by one or more classes.
- In an activity diagram partitioned into swimlanes, every activity belongs to exactly one swimlane, but transitions may cross lanes.



Swimlanes

Object Flow

- Objects may be involved in the flow of control associated with an activity diagram.
- We can specify the things that are involved in an activity diagram by placing these objects in the diagram, connected using a dependency to the activity or transition that creates, destroys, or modifies them.
- This use of dependency relationships and objects is called an object flow because it represents the participation of an object in a flow of control.
- We can also show how its role, state and attribute values change.
- We represent the state of an object by naming its state in brackets below the object's name.
- Similarly, We can represent the value of an object's attributes by rendering them in a compartment below the object's name.



Object Flow

Common Uses

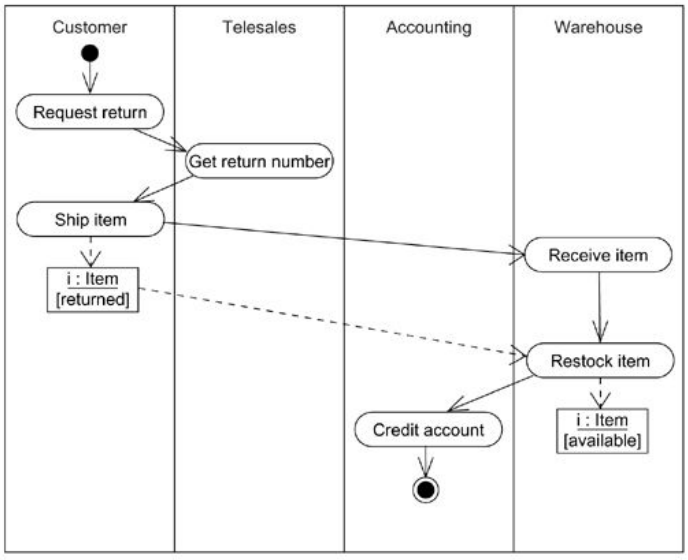
- We use activity diagrams to model the dynamic aspects of a system
- These dynamic aspects may involve the activity of any kind of abstraction in any view of a system's architecture, including classes, interfaces, components, and nodes.
- When you model the dynamic aspects of a system, we'll typically use activity diagrams in two ways.
    - To model a workflow
    - To model an operation

Common Modeling Techniques

Modeling a Workflow

- No software-intensive system exists in isolation; there's always some context in which a system lives, and that context always encompasses actors that interact with the system.
- Especially for mission critical, enterprise software, you'll find automated systems working in the context of higher-level business processes.
- These business processes are kinds of workflows because they represent the flow of work and objects through the business.

- To model a workflow,
    - Establish a focus for the workflow. For nontrivial systems, it's impossible to show all interesting workflows in one diagram.
    - Select the business objects that have the high-level responsibilities for parts of the overall workflow. These may be real things from the vocabulary of the system, or they may be more abstract. In either case, create a swimlane for each important business object.
    - Identify the preconditions of the workflow's initial state and the postconditions of the workflow's final state. This is important in helping you model the boundaries of the workflow.
    - Beginning at the workflow's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
    - For complicated actions, or for sets of actions that appear multiple times, collapse these into activity states, and provide a separate activity diagram that expands on each.
    - Render the transitions that connect these activity and action states. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
    - If there are important objects that are involved in the workflow, render them in the activity diagram, as well. Show their changing values and state as necessary to communicate the intent of the object flow.



Modeling a Workflow

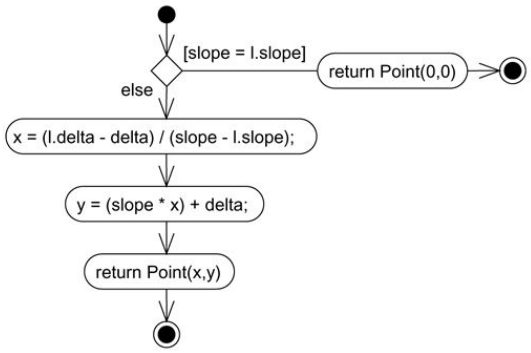
Modeling an Operation

- An activity diagram can be attached to any modeling element for the purpose of visualizing, specifying, constructing, and documenting that element's behavior.
- You can attach activity diagrams to classes, interfaces, components, nodes, use cases, and collaborations.
- The most common element to which you'll attach an activity diagram is an operation.
- An activity diagram is simply a flowchart of an operation's actions.
- An activity diagram's primary advantage is that all the elements in the diagram are semantically tied to a rich underlying model.
- To model an operation,
    - Collect the abstractions that are involved in this operation. This includes the operation's parameters (including its return type, if any), the attributes of the enclosing class, and certain neighboring classes.
    - Identify the preconditions at the operation's initial state and the postconditions at the operation's final state. Also identify any invariants of the enclosing class that must hold during the execution of the operation.

- Beginning at the operation's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.

- Use branching as necessary to specify conditional paths and iteration.

- Only if this operation is owned by an active class, use forking and joining as necessary to specif



Modeling an Operation

Forward and Reverse Engineering

- **Forward engineering** (the creation of code from a model) is possible for activity diagrams, especially if the context of the diagram is an operation.
- For example, using the previous activity diagram, a forward engineering tool could generate the following C++ code for the operation intersection.

```
Point Line::intersection (l : Line) {
  if (slope == l.slope) return Point(0,0);
  int x = (l.delta - delta) / (slope - l.slope);
  int y = (slope * x) + delta;
  return Point(x, y);
  }
```

- **Reverse engineering** (the creation of a model from code) is also possible for activity diagrams, especially if the context of the code is the body of an operation.
- In particular, the previous diagram could have been generated from the implementation of the class Line.